



SOUPS: A Variable Ordering Metric for the Saturation Algorithm

Benjamin Smith  Gianfranco Ciardo 
 Department of Computer Science, Iowa State University
 Ames, IA 50011, USA
 {bensmith@iastate.edu, ciardo@iastate.edu}

Abstract—Multivalued decision diagrams are an excellent technique to study the behavior of discrete-state systems such as Petri nets, but their variable order (mapping places to MDD levels) greatly affects efficiency, and finding an optimal order even just to encode a given set is NP-hard. In state-space generation, the situation is even worse, since the set of markings to be encoded keeps evolving and is known only at the end.

Previous heuristics to improve the efficiency of the saturation algorithm often used in state-space generation seek a variable order minimizing a simple function of the Petri net, such as the sum over each transition of the top variable position (SOT) or variable span (SOS). This, too, is NP-hard, so we cannot compute orders that minimize SOT or SOS in most cases but, even if we could, it would have limited effectiveness. For example, SOT and SOS can be led astray by multiple copies of a transition (giving more weight to it), or transitions with equal inputs and outputs (giving weight to transitions that should be ignored).

These anomalies inspired us to define SOUPS, a new heuristic that only takes into account the *unique* and *productive* portion of each transition. The SOUPS metric can be easily computed, allowing us to use it in standard search techniques like simulated annealing to find good orders.

Experiments show that SOUPS is a much better proxy for the quantities we really hope to improve, the memory and time for MDD manipulation during state-space generation.

I. INTRODUCTION

The saturation algorithm in its various forms represents the state of the art mechanism to generate the reachable state space for a system. Given a set of variables, the saturation algorithm requires an order for those variables prior to execution. Even for simple systems the choice of order can lead to dramatic differences in time and memory, and in the worst cases often stretches beyond the capacity of our machines and patience.

Numerous algorithms producing orders for use by saturation have been moderately successful in some but not all cases. Usually, these algorithms stem from those used for graph layout problems and do not take into account the special circumstances found using saturation. Whereas these graph layout problems of interest are NP-complete, it is unwise to spend computation optimizing them when they so often disagree with results in practice. We present a metric approach for comparing orders, with the goal of efficiently estimating which order will offer lower computation cost using saturation. The metric can be applied to any order making it possible to choose between orders derived from any source or algorithm. Further, the metric's efficient computation allows it to serve as the objective function for any heuristic search algorithm.

State of the art implementations of saturation use many techniques to improve performance, making it difficult to compare tools and further complicating our primary goal of tackling the ordering problem. We aim to make our findings repeatable, and as applicable as possible. To this end, we make available an implementation of saturation with minimal divergence from its modern incarnation [7]. The results presented demonstrating that our ordering metric correlates with saturation cost can be independently verified, and similar results can be expected when used with most implementations.

The rest of this paper is organized as follows. Section II summarizes the saturation algorithm, its use of decision diagrams, how we measure performance, and how variable ordering can improve results. Section III defines the SOUPS metric, explains how it relates to the reduction of costly operations performed by saturation, and includes a short proof that SOUPS is NP-hard. Section IV explains an efficient approach to calculate the SOUPS metric, and how it can be applied to searching. Section V presents experimental results demonstrating that SOUPS correlates with saturation cost, and that SOUPS is a more reliable metric. Section VI summarizes our findings.

II. BACKGROUND

This section recalls key concepts about Petri nets, decision diagrams, the saturation algorithm, and variable ordering.

A. Petri nets

A *Petri net* is a tuple $(\mathcal{P}, \mathcal{T}, \mathbf{F}^-, \mathbf{F}^+, \mathbf{m}^{init})$, where \mathcal{P} and \mathcal{T} are sets of *places* and *transitions*, with $\mathcal{P} \cap \mathcal{T} = \emptyset$ and $\mathcal{P} \cup \mathcal{T} \neq \emptyset$, $\mathbf{F}^-: \mathcal{P} \times \mathcal{T} \rightarrow \mathbb{N}$ and $\mathbf{F}^+: \mathcal{P} \times \mathcal{T} \rightarrow \mathbb{N}$ are *incidence matrices* specifying the cardinalities of the *input* and *output* arcs between p and t , and $\mathbf{m}^{init} \in \mathbb{N}^{\mathcal{P}}$ is the *initial marking* (a marking assigns a number of *tokens* \mathbf{m}_p to each $p \in \mathcal{P}$).

The dynamics of the net are governed by the *enabling rule* (transition t is enabled in marking \mathbf{m} if, for each place p , the input arc is satisfied, $\mathbf{m}_p \geq \mathbf{F}_{p,t}^-$) and by the *firing rule* (transition t enabled in marking \mathbf{m} may fire, leading to marking \mathbf{n} , where, for each place p , $\mathbf{n}_p = \mathbf{m}_p - \mathbf{F}_{p,t}^- + \mathbf{F}_{p,t}^+$). For general discrete-state formalisms, the *next-state function* \mathcal{N}_t for event t returns a set of states when applied to a single state \mathbf{m} , thus we write $\mathcal{N}_t(\mathbf{m}) = \{\mathbf{n}\}$, so that $\mathcal{N}_t(\mathbf{m}) = \emptyset$ means that t is not enabled in marking \mathbf{m} . We let $\mathcal{N} = \bigcup_{t \in \mathcal{T}} \mathcal{N}_t$ be the

overall next-state function, and extend our notation to sets of markings \mathcal{X} , i.e., $\mathcal{N}(\mathcal{X}) = \{\mathbf{n} : \exists \mathbf{m} \in \mathcal{X}, \mathbf{n} \in \mathcal{N}(\mathbf{m})\}$.

The *reachability set* describes the markings reachable from \mathbf{m}^{init} through firing sequences, $\mathcal{S}_{reach} = \{\mathbf{m}^{init}\} \cup \mathcal{N}(\mathbf{m}^{init}) \cup \mathcal{N}(\mathcal{N}(\mathbf{m}^{init})) \cup \dots = \mathcal{N}^*(\mathbf{m}^{init})$, and we consider only the case where it is finite, which implies that the number of tokens in each place p is bounded by some value b_p .

B. Ordered multi-valued decision diagrams

An L -level MDD [12] over a finite set $\mathcal{S} = \mathcal{S}_1 \times \dots \times \mathcal{S}_L$ is an acyclic directed edge-labeled level graph with *terminal* nodes 0 and 1, at level 0, while each *nonterminal* node p is at some level $p.lvl = k \in \{1, \dots, L\}$, and, for $i_k \in \mathcal{S}_k$, has an outgoing edge labeled with i_k and pointing to a child $p[i_k]$ at level $p[i_k].lvl < k$ (this is to enforce the “ordered” property). Without loss of generality, we can assume that each \mathcal{S}_k is of the form $\{0, \dots, b_k\}$, for some $b_k \in \mathbb{N}$.

MDD node p at level k encodes function $f_p : \mathcal{S} \rightarrow \mathbb{B}$, recursively defined by $f_p(i_1, \dots, i_L) = f_{p[i_k]}(i_1, \dots, i_L)$, with base case $f_p(i_1, \dots, i_L) = p$ when $k = 0$. Interpreting f_p as an indicator function, p encodes set $\mathcal{X}_p = \{\mathbf{i} : f_p(\mathbf{i}) = 1\} \subseteq \mathcal{S}$. To encode relations over \mathcal{S} , we use $2L$ -level MDDs over $(\mathcal{S}_1 \times \mathcal{S}_1) \times \dots \times (\mathcal{S}_L \times \mathcal{S}_L)$, where the first set in each pair corresponds to a “from”, or “unprimed”, local state and the second set corresponds to a “to”, or “primed”, local state.

We also say that \mathcal{S}_k , for $1 \leq k \leq L$, describes the possible values of *domain variable* x_k , so an MDD maps x_1, \dots, x_L to the *range variable* x_0 , taking values over $\mathcal{S}_0 = \{0, 1\}$. The *variable order* used to match the L domain variables to the L levels may greatly affect the size of the MDD encoding a given function, and finding an optimal order is NP-hard [4].

MDDs are *canonical*, i.e., the MDD encoding a given function f with a particular variable order is unique, if we forbid *duplicate* nodes (there cannot be distinct nodes p and q with $p.lvl = q.lvl = k$ and $p[i_k] = q[i_k]$ for all $i_k \in \mathcal{S}_k$) and either eliminate all *redundant* nodes (a node p must have at least two distinct children) so that edges skip as many levels of *don’t care* variables as possible, or we keep all redundant nodes, so that no edge skips levels. The two resulting canonical forms are called *fully-reduced* and *quasi-reduced*, respectively.

When encoding relations, we employ a third canonical form, *fully-identity-reduced* [7], where unprimed levels are fully-reduced but primed levels are identity-reduced, meaning that the semantic of an edge skipping over a primed level k' is that the value of x'_k is the same as that of x_k . This form can result in much more compact encoding of next-state functions if events depend and affect only on a small subset of state variables, as is the case for most transitions in a Petri net, since their enabling depends only on the input places and their firing changes only the input and output places.

Figure 1 shows examples of these reduction forms assuming $\mathcal{S}_k = \{0, 1, 2\}$ and omitting edges going to terminal node 0. In the left panel, we see how the redundant node at level k is present in the quasi-reduced form but not in the fully-reduced form. In the center panel, the *singular node* at level k' is pointed to by two edges from the node at level k when both

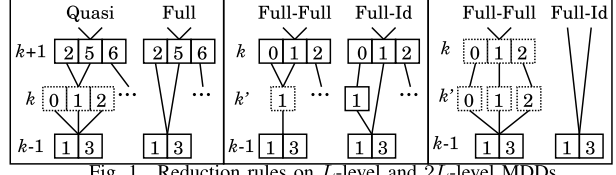


Fig. 1. Reduction rules on L -level and $2L$ -level MDDs.

unprimed and primed levels are fully-reduced, but only by the 0-edge when primed levels are identity-reduced. In the right panel, all nodes at level k' are eliminated if primed levels are identity-reduced, and this in turn makes the node at level k redundant (all its edges now point to the node at level $k - 1$), so is itself eliminated since unprimed levels are fully-reduced. This *identity pattern* describes the common situation where x_k neither affects the enabling of the transition encoded by the $2L$ -level MDD, nor it is affected by its firing, and is key to recognizing *locality* in the MDD encoding a transition relation.

C. The saturation algorithm

Given a Petri net with $|\mathcal{P}| = L$, we can map its places to the variables of an L -level MDD and build an initial MDD encoding the set $\mathcal{X}_0 = \{\mathbf{m}^{init}\}$ as well as an MDD encoding \mathcal{N}_t , for each $t \in \mathcal{T}$. Then, we can *symbolically* generate \mathcal{X}_{reach} by building the MDD encoding \mathcal{N} , i.e., computing the *union* of the MDDs encoding the relations \mathcal{N}_t , and then computing the sequence of assignments $\mathcal{X}_{n+1} \leftarrow \mathcal{X}_n \cup \mathcal{N}(\mathcal{X}_n)$, each of them requiring a union and a *relational product*, until we reach a fixpoint. This is possible only if all places are bounded, but we do not need to know their bounds a priori, we can instead incrementally build \mathcal{N}_t (and update \mathcal{N}) *on the fly* [6]. We do not even need to build \mathcal{N} , we can instead apply each \mathcal{N}_t at each iteration, and this often is more efficient because the MDD encoding each \mathcal{N}_t is usually small if fully-identity-reduced, while the MDD encoding the overall \mathcal{N} might explode in size. However, the fundamental problem of this simple approach is its *breadth-first* flavor: \mathcal{X}_n contains exactly all reachable markings at distance up to n from \mathbf{m}^{init} , and the MDD encoding this set is often huge even if the MDD encoding the fixpoint \mathcal{X}_{reach} might not be. This is a well-known problem: the *peak* MDD size is the critical factor for memory (and time) performance, not the *final* MDD size.

The *saturation* algorithm [5] attempts to reduce peak MDD size through *chaotic-style* nested iterations. When \mathcal{N}_t is encoded with a fully-identity-reduced MDD, we can easily determine the levels $Top(t)$ and $Bot(t)$ enclosing variables affected by or affecting transition t , so that all variables above $Top(t)$ or below $Bot(t)$ are neither changed by firing t , nor they can disable it. Then, saturation repeatedly applies the next-state functions \mathcal{N}_t starting from the ones with the lowest Top , and immediately applying lower ones whenever new lower MDD nodes are created. An MDD node at level k is said to be saturated if it encodes a fixpoint with respect to all \mathcal{N}_t with $Top(t) \leq k$, and the fundamental property of the saturation algorithm is that all descendant nodes from the node being saturated are guaranteed to be saturated already. Experimental results have shown that saturation can have peak

memory consumption and runtimes thousands of times smaller than breadth-first, so it is now the preferred algorithm for MDD-based state-space generation of discrete-state models having asynchronous events.

The size of the MDD encoding \mathcal{X}_{reach} is obviously a lower bound on the size of the peak MDD for saturation.

D. Measuring the cost of a saturation run

Without assuming specific implementation details, the basic operations needed by saturation are creation of nodes, cache lookups, and computing unions and relational products. Both time and memory are critical when generating a large reachability set but, with MDDs, the two are usually tightly correlated in the sense that the number of nodes being manipulated determines both the storage and the runtime requirements¹.

Each call to saturation, firing, and union creates a node, so node creation is a reliable and countable indicator of overall work being done. Every node created during execution must be checked into the unique table. Previous work often seeks to use “peak nodes” during a saturation run as a measure of memory usage, the motivation for this being that the computation will in principle fail if it needs to store more nodes than can fit in the computer’s memory. In practice, though, the implementation resorts to some form of garbage collection to delete disconnected nodes. The two extreme approaches are then to store every node created indefinitely i.e., never run garbage collection, or to delete every node as soon as it becomes disconnected. The latter “pessimistic” approach leads to the smallest peak size but has terrible performance in practice, not just because of its high garbage collection overhead, but also because deleted nodes may have to be recreated later, if a different path in the MDD needs to encode the same set of submarkings. The former “optimistic” approach of preserving all created nodes leads to the highest peak size but is closer to most implementations, which try to avoid recomputations and maximize cache hits as much as possible; it also results in the best runtime, as long as its memory requirements are not excessive. We choose to track the number of MDD nodes actually inserted in the unique table: if no garbage collection is ever invoked, this is exactly the number achieved with the optimistic approach; otherwise, it is an upper bound on that number, since a node may be created and inserted in the unique table, then become disconnected and deleted by a garbage collection pass, then created and inserted again later.

E. Variable ordering heuristics

Strategies for coping with the variable ordering problem include searching the space of orders to improve a *metric*,

¹We observe that, unlike BDDs where each node has exactly two edges, MDDs are best stored in some sparse format where only edges not pointing to terminal 0 occupy memory, so that a node at level k may have to store up to $|\mathcal{S}_k|$ of them. The number of edges is thus a more accurate measure of MDD memory usage than the number of nodes, but in practice we can simply track the latter, as long as the individual domains \mathcal{S}_k are reasonably small. More importantly, we focus on the effect of different variable orders on the time and memory requirements when running saturation on a given model, and the ratio of edges to nodes is quite consistent for these experiments.

or using a specialized polynomial algorithm to produce an order. While we employ the searching approach, existing polynomial algorithms often produce results good enough to solve many problems. When these algorithms fail to produce orders that allow saturation to complete, the only alternative is to adjust their parameters and try saturation again, essentially performing a search limited by the flexibility of the algorithm. With a sufficiently accurate metric, the metric can serve as a substitute for running saturation, saving time.

Most successful polynomial algorithms were conceived for different problems. The FORCE algorithm was intended as a simpler alternative to algorithms reducing the size of BDDs, and is defined for SAT problems [3]. The idea of obtaining a layout by assigning forces to movable components has applications in diverse settings, and more flexible approaches have been around even longer [10]. This flexibility is necessary, as we will explore factors in addition to keeping connected places as close as possible. The Cuthill-McKee [8] and Sloan [19] algorithms target permutations to improve numerical computations with sparse matrices. Other than Noack [15], most polynomial algorithms target problems only loosely related to Petri net variable ordering [9][17]. The general graph layout problem fails to consider several performance-critical phenomena exploited by saturation, and observing when these occur can improve these polynomial algorithms.

The variable order affects the cost of saturation in several ways. Saturation benefits if a transition t to be fired has a low $Top(t)$, since a larger portion of the decision diagram above $Top(t)$ remains unchanged. It also benefits if the range from $Top(t)$ to $Bot(t)$ is narrow, since no relational product is performed on nodes at levels below $Bot(t)$. These two observations lead to the development of the first heuristics for variable order specifically targeted toward saturation [18], attempting to find an order that minimizes the “sum-of-tops” (SOT), $\sum_{t \in \mathcal{T}} Top(t)$, or the “sum-of-spans” (SOS), $\sum_{t \in \mathcal{T}} Top(t) - Bot(t) + 1$, respectively. Unfortunately, both computations have been shown to be NP-hard, so in practice one can only hope to find orders with “low” SOT or SOS.

Of course, any good implementation of the saturation algorithm employs caches for saturation, firing, and union operations, but this further complicates an *a priori* assessment of the actual cost of saturating nodes at a given level. Furthermore, a careful cache implementation based on *shared* MDD nodes also implies that firing distinct but equivalent transitions (or portions of transitions) also results in cache hits. This means that not only it is impractical to compute variable orders that optimize SOT or SOS, but also that, even if we were able to do so, these orders would not necessarily be optimal for saturation: unfortunately, SOT and SOS are just proxies for the time and memory requirements of a saturation run.

Since even just finding an optimal order to encode the final result of saturation is NP-hard, it is not surprising that the proposed heuristics for saturation are NP-hard and the optimization problems they imply can be only partially addressed with polynomial-time heuristics, which are in turn sensitive to the initial variable order. In other words, different “good”

(according to a given heuristic) orders are generally obtained if we start the search algorithm from different initial orders. This also holds for polynomial time heuristic algorithms which rely on randomness or unspecified criteria to break ties in their calculations. In conclusion, no known algorithm provides saturation with a clear choice for a variable order.

III. THE SOUPS METRIC

SOT and SOS are reasonable metrics, but fail to reflect that not all transition spans have the same firing costs. Unlike SOT, SOS is not concerned with nodes below $Bot(t)$ since no calls to fire t are issued on them; SOS is thus equivalent to SOT with a discount applied to the portion of a span below the bottom variable. We stress that a good SOS order does not always produce better results than a good SOT order. For example, a variable order and its reverse have the same SOS score, but may result in greatly different saturation performance: this confirms that there is a directional component in the cost of firing a transition, which SOS ignores but SOT does not.

A. Definition

Our definitions, and our implementation, rely on the notion of *arc-pair*. The arcs of a Petri net are described in terms of two incidence matrices \mathbf{F}^- and \mathbf{F}^+ specifying cardinalities of the arcs that subtract or add tokens, respectively. An arc-pair $A_{p,t}$ is a tuple containing both cardinalities for a given $p \in \mathcal{P}$ and $t \in \mathcal{T}$. This allows us to combine the \mathbf{F}^- and \mathbf{F}^+ matrices in a way that does not discard information. An arc-pair $A_{p,t} = (0, 0)$ means that there is no arc between place p and transition t , but it is enough to have $A_{p,t} = (n, n)$ for any $n \in \mathbb{N}$ to ensure that transition t has no net effect on place p . We arrange these arc-pairs into sequences $Seq(t)$ for each transition $t \in \mathcal{T}$ according to the variable order, where the lowest level starts the sequence and the highest level ends it. Sequence $Seq(t)$ contains all arc-pairs for transition t , including those that are $(0, 0)$. A prefix of $Seq(t)$ of length $Top(t)$ contains all of the arc-pairs necessary to calculate the cost of a transition t , as it contains the cardinalities of all arcs connected to t . We call this sequence $TopSeq(t)$, so $SOT = \sum_{t \in \mathcal{T}} Length(TopSeq(t))$. A suffix of $TopSeq(t)$ starting with the arc-pair at level $Bot(t)$ has length equal to the span of transition t . If we call this sequence $SpanSeq(t)$, then $SOS = \sum_{t \in \mathcal{T}} Length(SpanSeq(t))$.

A transition that removes and adds an equal number of tokens to each place does not result in any new marking: it is *non-productive*. This is of course a useless transition, thus it would not be present in practical nets. However, from the point of view of the recursive firing operation, a transition where the lowermost levels are all non-productive is no different from a transition which is entirely non-productive. The result returned is the same, and in both cases the corresponding places remain unchanged. For this reason, we consider these lower non-productive spans less costly, and discount them when considering the contribution of t . This leads to a new metric, the sum-of-productive-spans (SOPS). For transition t , productive substring $Prod(t)$ can be defined as the suffix of $TopSeq(t)$ starting with the first productive arc-pair,

$(\mathbf{F}_{p,t}^-, \mathbf{F}_{p,t}^+)$ s.t. $\mathbf{F}_{p,t}^- \neq \mathbf{F}_{p,t}^+$. The metric can then be calculated as $SOPS = \sum_{t \in \mathcal{T}} Length(Prod(t))$.

Until now the cost of firing a transition has ignored the other transitions in the model, treating each one in isolation. Imagine now a Petri net with multiple transitions having exactly the same input and output arcs. These transitions affect the marking in the same way, thus all but one are redundant. Again, no practical model would have such repeated transitions, yet the definition of SOT and SOS, at least as originally given, would count the contribution of each of these identical transitions multiple times, skewing the value of the metric so that orders that reduce the top or span of these multiple copies would be preferred, at the expense of other transitions. Of course, one could check for and eliminate duplicate transitions by comparing the arc-pair sequences, and certainly a good MDD-based implementation of the relational product would automatically result in cache hits when they are present. However, as above, while multiple transitions with the exact same effect might rarely be present, it is not uncommon to have transitions that share portions of their effect on the marking, and we focus on the case where this happens at the last few bottom levels, since only in these cases firing cache hits are possible. Thus, the contribution of these levels to the metric should be reduced or even ignored. We wish to discount all occurrences where a cache hit is possible, counting only those spans where the upper structure of a transition is unique.

This leads to a new metric, the sum-of-unique-spans (SOUS). SOUS is the number of unique prefixes of $TopSeq(t)$ for all t such that the prefix contains at least one non-empty arc-pair. Let $LowArc(t)$ be the lowest non-empty arc-pair for transition t , then

$$SOUS = \left| \bigcup_{t \in \mathcal{T}} \{s = Prefix(TopSeq(t)) : LowArc(t) \in s\} \right|$$

The final combined metric sum-of-unique productive-spans (SOUPS) considers only those spans that are both unique and productive. The definition of the SOUPS metric is similar to that for SOUS, except that it counts unique spans only if they are also productive. SOUPS is the number of unique prefixes of $TopSeq(t)$ for all t such that the prefix contains at least one productive arc-pair. Let $LowProd(t)$ be the lowest productive arc-pair for transition t , then

$$SOUPS = \left| \bigcup_{t \in \mathcal{T}} \{s = Prefix(TopSeq(t)) : LowProd(t) \in s\} \right|$$

Section IV-B describes an efficient way to compute this metric.

B. Complexity

It is easy to see that finding an order with an optimal SOUPS (or SOPS, or SOUS) score is NP-hard. Assume we had a polynomial algorithm to find an optimal SOUPS score. Then, given any Petri net, we could change it so that all arcs have different cardinalities. This new net would then have the same SOS as the original one, but there would be neither non-productive nor non-unique spans, thus running the hypothetical

polynomial SOUPS algorithm on the new net would give an optimal order for SOS as well, which is known to be NP-hard.

C. Example

It is important to understand the reasons why SOS and SOT perform poorly on some models. One such anomaly is the Eratosthenes Petri net from the MCC competition [2], shown in Figure 2, where every transition contains one non-productive arc-pair and one arc removing a single token from a place. If an order is such that all transitions have the place from which the token is removed as their top, then the entire portion of each span below the top is non-productive. It is obvious that any such order has an optimal SOUPS score, while its SOS or SOT score would be poor.

We visualize variable orders for Petri nets as in Figure 3. Each grid square represents an arc-pair with output cardinalities on the left and input cardinalities on the right, with the (0,0) pair omitted; black squares indicate the productive and unique portions of a span, and gray squares indicate the non-productive or non-unique portion. Figure 3 illustrates the extreme difference between SOS and SOUPS for the Eratosthenes model, since the *area* of the black squares equals the SOUPS value, while that of the black or gray squares equals the (much larger) SOS value.

It turns out to be quite easy to generate optimal SOUPS orders for this model in linear time using a breadth-first search, and these optimal SOUPS orders indeed result in very good runtimes, taking a fraction of a second to complete saturation even for the largest instances of this model. Orders generated while seeking to improve other metrics do not produce similar results, and often cannot complete at all. In other words, this is a model where focusing on the wrong factors is disastrous.

In their survey of graph layout problems, the authors of [9] enumerated many classes of graphs with polynomial time algorithms for finding optimal orders under various metrics, even though the general problem is NP-hard. This is the case with Eratosthenes and the SOUPS metric, and likely many others, where finding an optimal SOUPS order via search is far more difficult. There exist relatively few, usually very small, cases where we have proved that an optimal SOUPS order is obtained, and in all of those cases the results have been quite good. Even though the metrics share similar motivations, a good SOUPS order can have a very different structure from those optimized for SOS or SOT. Many algorithms used to generate orders for use in saturation have been successful, and often compute good SOUPS orders merely by accident. These approaches may benefit from modifications that more purposefully target good SOUPS orders.

IV. EFFICIENT COMPUTATION OF THE SOUPS METRIC

For the SOUPS metric to be useful, it should be efficiently computable. A naïve implementation of any metric could be wasteful, but the checking for uniqueness as we defined it could be especially costly. We seek to count only those spans which are unique among the transitions, which suggests the need to maintain a set of previously encountered spans.

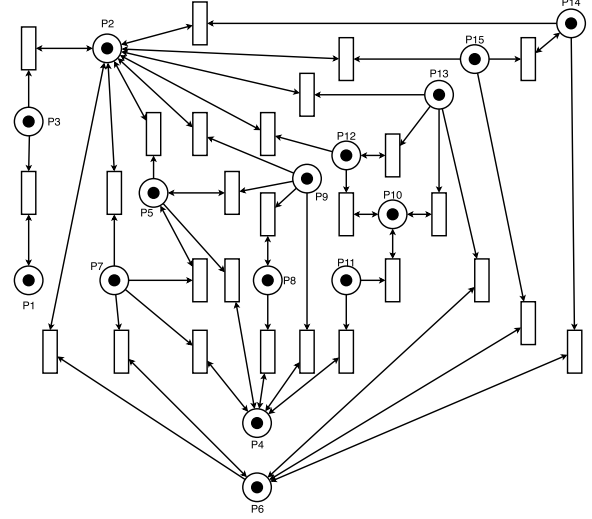


Fig. 2. Petri net diagram for Eratosthenes-20

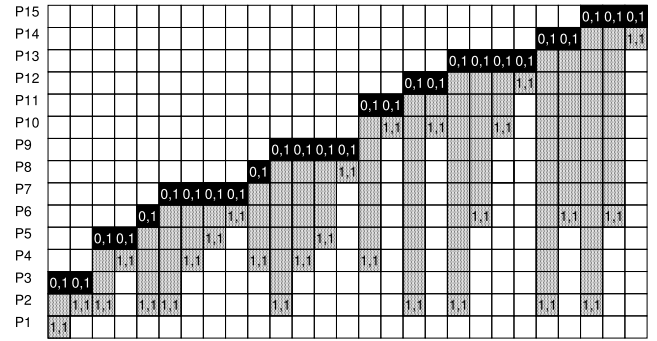


Fig. 3. An optimal SOUPS order for Eratosthenes-20

Implemented in this way, the memory required to store this set would grow when an order has many large unique spans, but this is in fact not necessary, and all the metrics we described can be calculated with memory requirements in the number of arcs in the Petri net. In addition to saving memory, time improvements are important as well, since they allow us to search more of the vast space of possible orders.

For every transition, we store its non-zero arc-pairs in a balanced binary search tree which maps the level associated with each place to the corresponding arc-pair. The ordered mapping provided by the binary tree allows for quickly determining the lowest and highest connected places, iterating according to the order, and for quickly updating the structure when modifying the order. Every lookup, deletion, and update can be done in logarithmic time, since the height of the tree is logarithmic with respect to the number of arcs connected to the transition. Since most transitions are connected to few places, the height of these trees is very small in practice.

The minimum and maximum entries of each binary tree correspond to the bottom and top for each transition respectively, making the calculation of SOS and SOT a simple matter of finding these values from the binary trees created for each transition. The productive span can be obtained by iterating

over each tree from the minimum to the maximum, obtaining the lowest level with a productive arc-pair.

A. Updating and searching

We find good orders for our metrics using simulated annealing, which attempts to improve upon some fitness metric by making slight changes [14]. Simulated annealing avoids the problem of getting stuck in local minimum by continuing to pursue changes, even when those changes do not improve upon their previous value. This approach has been shown to be among the most effective for similar ordering problems, especially when the search time is not a concern [16].

To use our metrics in the inner loop of a search heuristic such as simulated annealing, we want to calculate a new value for them after a simple change without having to recalculate the entire score or rebuild data structures. Specifically, we consider obtaining one order from another by swapping the locations of two places in the order. These swaps can be performed on the binary trees for each transition by removing the entries for the old locations and inserting them in their new places. This process can be improved by keeping track of which transitions are connected to each place, so that only those transitions affected by the swap need to be updated.

We observe that all of the metrics described in this paper grow monotonically while they are being calculated, thus we can stop the calculation of a metric when the current sum exceeds its previous value, allowing particularly bad updates to be abandoned early and avoiding the expense of finding the full sum [13]. We use this point of abandonment as an estimate of the quality of the updated order, rather than computing the full value and measuring the change. A very early abandonment indicates an update that should have a lower probability of being followed further, while an abandonment that has calculated nearly the entire metric is more promising. Of course, a full calculation of the metric without abandonment occurs when an improvement is found, and these updates are always followed.

B. SOUPS calculation

The uniqueness property must consider multiple transitions, so it cannot be directly measured from each transition in isolation, as with the other metrics. Sorting the transitions can put them in an order such that transitions adjacent in the order have the lowest possible point of differentiation. At this level and above a transition is unique compared with all of its predecessors. Determining unique span requires only comparison with one other transition if the transitions are in this sorted order.

The first aspect of the sort is according to the bottom, since having the same bottom is a necessary condition for non-uniqueness. Sorting the lowest bottoms first also aids our early abandonment approach, by considering larger spans first, which may let us discover sooner that an order has low quality. Among transitions with the same bottom level, a comparison between two transitions is based on their lowest differing arc-pair. Any consistent comparison between differing arc-pairs

leads to the same uniqueness result. Once the transitions are in this sorted order, the lowest level at which a transition differs from its predecessor indicates that at this level and above only, the span is counted as unique, since the equal portion of the span must have already been counted for an earlier transition.

This sorting of transitions potentially requires comparisons where each arc in a pair of transitions is considered. As with the other metrics, swapping a pair of levels may not affect every transition, and will not always result in a change to the sorted order of transitions. Taking advantage of this observation could lead to further speed improvements, as avoiding sorting the entire collection of transitions after each update may be faster in practice. We caution against using a sorting algorithm such as a poorly implemented quicksort where the worst performance occurs when input is nearly sorted, as this is often the case when sorting after making a small change to a variable order.

V. RESULTS

To evaluate our heuristic, we implemented a saturation-based algorithm to generate the state space of a Petri net. This prototype is freely available [1] so anyone can readily replicate or extend our results. The tool reads a model in PNML format [11], then derives a variable order that saturation uses to generate the state space while collecting and reporting performance measurements.

We compare the SOT, SOS, SOPS, SOUS, and SOUPS heuristics on models from the Model Checking Competition (MCC) [2], which provides a diverse collection of Petri nets. Results from several tools that use some variant of saturation are available on the MCC website as well. While we do not directly compare our results with past competitors in the MCC state-space-generation category, we can state that good SOUPS orders enabled us to generate the reachability set for instances not previously completed by any tool in competition. This suggests that saturation using good SOUPS orders would be a highly competitive entry in MCC.

For each model, we run a simulated annealing search to find good SOT, SOS, SOPS, SOUS, or SOUPS orders starting from 100 random orders. This provides us with up to 500 distinct orders per model (it may be fewer than 500, since the sets of orders derived for the individual metrics may not be disjoint, this is more likely to happen for models with few variables, where the total number of orders is itself not that large, or when the heuristics agree on what orders are good: for example, SOS, SOPS, SOUS, and SOUPS agree on the score when all transitions are fully productive and unique).

Since we run the simulated annealing search for 50,000 iterations, we calculate each metric this many times, producing one order to use with saturation. This means that we have explored as many as $5 \times 100 \times 50,000 = 25$ million orders for each model, but even this is often only a tiny fraction of the possible orders. To ensure reliable and insightful results, we focus on model instances where we are able to complete saturation using these orders and provide measurements we can analyze. We wish to study the effectiveness of our metrics,

SOUPS in particular, and focusing on a large but manageable set of reasonably good orders narrows the search space significantly, while providing more insight than even a much larger set of randomly generated orders would.

A. Evaluation criteria

The performance of saturation can be best measured using runtime and memory usage, but these quantities often do not provide fine-grained information about where the algorithm consumes resources. Runtimes can be very small, often fractions of a second, in which case they fail to illustrate meaningful information which can only be pinpointed by counting individual operations. Of course, they are also highly dependent on the quality of the implementation, and the hardware environment can introduce unpredictable variance. Our prototype implementation of saturation collects *counts* of every major operation, including the number of nodes and edges created, allowing confirmation that these counts are strongly correlated with runtime for larger more time-consuming models, while having the advantage of being largely independent of the specific implementation and hardware environment.

We restrict our analysis to the count of nodes created during execution. This measurement provides a single value to be used as a proxy for overall computation cost, as each call to the saturate, relational product, and union operations creates and checks a node into the unique table.

SOUPS takes advantage of the circumstances reducing the number of required uncached calls to the fire operation, and this is clearly apparent in our experiments, since cache hits do not result in a new node being checked in. Observing this count of created nodes allows us to confirm that performance improvements appear as a consequence of reducing specific operations. The cost savings from non-productive firings is estimated by counting the number of times a union returns one of its arguments as a result, indicating that firing created new sets but not additional states. Unions with this property occur while executing any order, but are more prevalent when non-productive regions are fired. As a whole, the collected data supports our claim that better values for the SOUPS metric result in reduced computation costs.

B. Experimental results

Figures 4, 5, 6, 7, and 8 show “normalized score” vs. “normalized node count” scatter plots for SOT, SOS, SOPS, SOUS, and SOUPS, respectively. For example, the SOT plot is obtained as follows. For each model X , define the highest (worst) SOT score $s(X)_{max}$ and the highest (worst) number of nodes created $n(X)_{max}$ when running saturation using the (up to) 500 orders for that model; then, define a normalized data point “ $s(X)_\pi/s(X)_{max}$, $n(X)_\pi/n(X)_{max}$ ”, where $s(X)_\pi$ and $n(X)_\pi$ are the score of order π and the number of nodes created by saturation using order π on model X ; the plot shows all the data points for all different orders of all models.

This normalization allows us to compare the five heuristics across different models, even if they may have very different node requirements. We stress that, while we maintain fairness

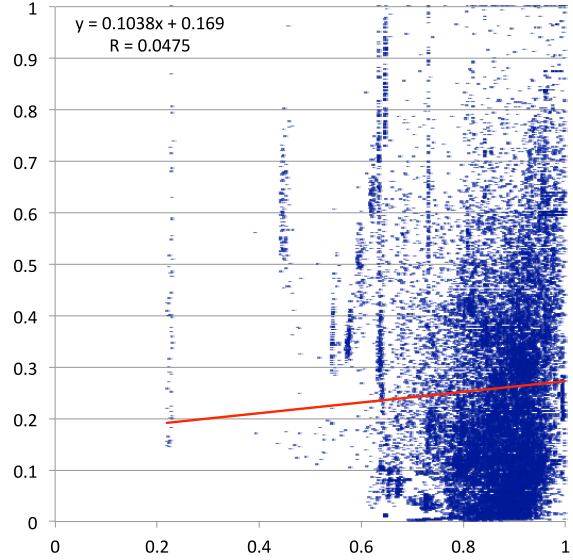


Fig. 4. Normalized SOT score (x) vs. node count (y).

by generating an equal number of orders deemed good by each of the five heuristics, each of these heuristics provides a score for any order, regardless of whether that order is deemed good or not by that heuristic. This seems an appropriate approach to pit the five heuristics against each other.

The plots show the correlation coefficient R , between the heuristic assessment of an order and its actual goodness in terms of created nodes.

- For SOT, the correlation is negligible. Particularly problematic is the large number of data points in the upper left corner, indicating orders that are deemed good but instead perform poorly. The plot also shows vertical clusters indicating the metric sometimes provides little to no information predicting node count.
- For SOS, the correlation is much better, the regression line clearly showing that higher scores *tend* to result in worse performance. However, the spread of the data points is still large and the correlation is weak (where 1.0 would be a perfect correlation).
- For SOPS, SOUS, and SOUPS, the situation is substantially better, with data points indicating a much clearer correlation. Particularly noticeable are the two clusters of data points on the lower left portions of the SOUS and SOUPS plots, suggesting that improving the score of an already good order tends to further improve performance. Of these three new heuristics, SOUPS is clearly the one with the best correlation, $R = 0.51457$.

A different way to approach a comparison between the five metrics is to consider how we would go about tackling a particular model. In practice, we could not afford nor we would be interested in generating 100 orders deemed good according to SOT, SOS, SOPS, SOUS, and SOUPS and then run saturation up to 500 times. Rather, we would have to choose one of these five heuristics, spend a reasonable amount of time to generate an order with a good score according to it,

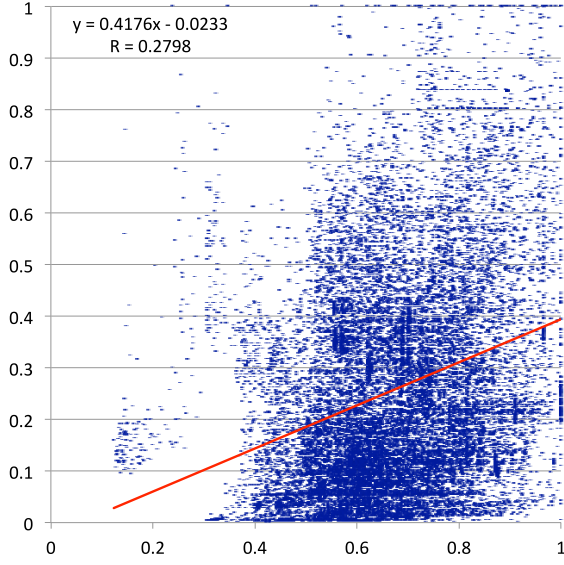


Fig. 5. Normalized SOS score (x) vs. node count (y).

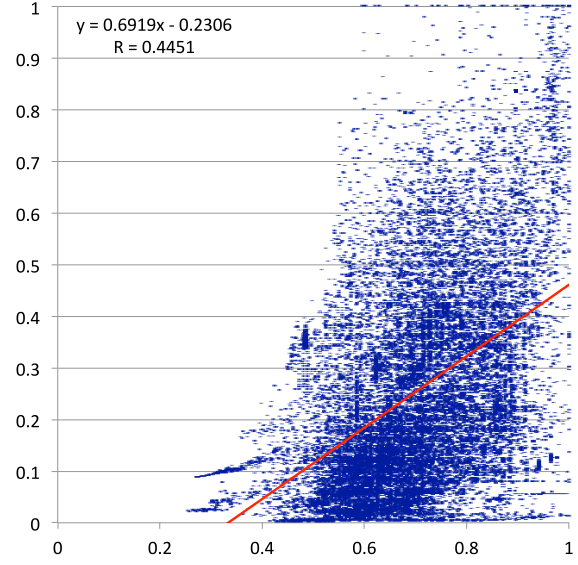


Fig. 7. Normalized SOUS score (x) vs. node count (y).

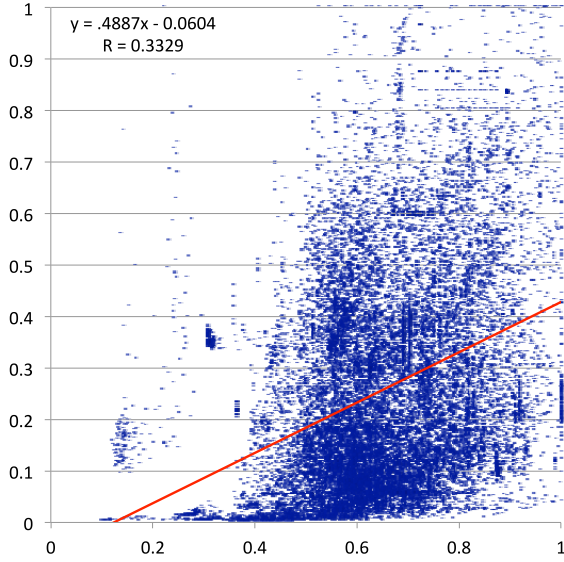


Fig. 6. Normalized SOPS score (x) vs. node count (y).

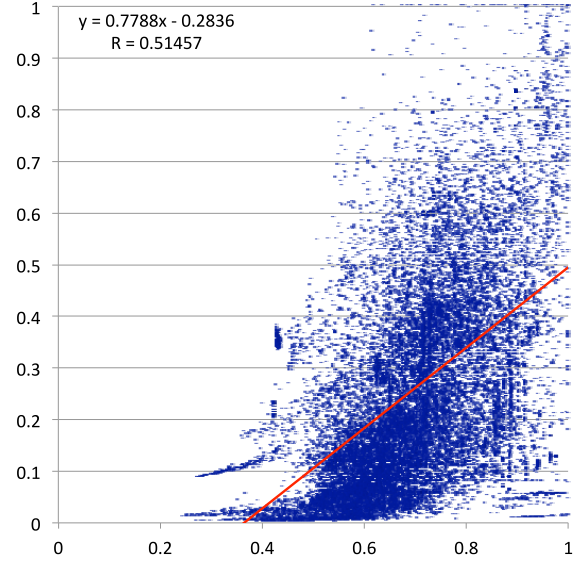


Fig. 8. Normalized SOUPS score (x) vs. node count (y).

and then run saturation once on the model. Multiple saturation runs using different orders would be required only if we keep being unable to complete the computation, and try a different order hoping that it will lead to a successful run.

Table I shows the same data used for the scatter plots, but organized differently. For each model we considered (the names are those from MCC, sometimes abbreviated to save space), we list the total number “Ord” of distinct orders, then, for each heuristic, we list, the correlation coefficient “ R ”, the minimum number of nodes created “Min”, the average number of nodes created “Avg”, and the maximum number of nodes created “Max”, computed only over the order or orders with the best score among all of those generated using simulated annealing, together with the number of such orders “Tie” (the

models are listed in decreasing order of “Avg” for SOUPS). This reflects what we would do in practice: we would choose one of the heuristics, say SOUPS, then run every ordering technique at our disposal, resulting in a set of orders, hopefully with good SOUPS scores. If only one of these orders has the minimum SOUPS score, then we run saturation using it; if, however, “Tie” orders have the lowest score, then we would randomly choose one of them and run saturation with it, in which case our performance may be as good as “Min” or as bad as “Max”, depending on how lucky we are, and it will be “Avg” on average.

Then, two useful comparisons can be made. First, when multiple orders tie for the lowest score according to a particular heuristic, the smaller the spread between “Min” and

“Max” is, the less dependent on luck we will be. Second, and more importantly, we can compare the “Min” (if we feel optimistic), “Max” (if we feel pessimistic), or “Avg” (maybe more realistically) values for SOT, SOS, SOPS, SOUS, and SOUPS, to determine which heuristic we might want to pick before tackling a model of which we know nothing about. Here, again, the choice is very clear. While there are a few models where SOT is better than SOUPS by a small factor (e.g., Circadian Clock), SOUPS is much better than SOT in the large majority of cases, often by a factor of 10 or more. The comparison between SOS and SOUPS also favors the latter, although not in as many models and not by as large factor, which is however not surprising, since SOUPS is essentially meant to be a refinement and an improvement over SOS.

In practice, we can conclude that SOUPS is our heuristic of choice if we have no other information about the model, although it is clearly not perfect, suggesting that there must be further factors influencing performance. In practice our search would likely include running every one of the polynomial algorithms at our disposal, and extensive experimentation on the best parameters for simulated annealing, with the hope that this approach would more often give the best result. As it stands, our basic implementation of simulated annealing sometimes finds the best order for a given metric while optimizing one of the others. Ideally the search should be capable of being run once, getting a consistent and usable result without a human manipulating parameters.

VI. CONCLUSIONS

We introduced SOUPS, a metric that can be effectively used to choose between variable orders prior to running saturation. As it is common practice to use one or more polynomial algorithms to generate several orders, SOUPS can be used to detect situations where one of these algorithms produces exceptionally good or bad results. As a general strategy, we suggest that the order with the best SOUPS score should be the first one tried on a new model when using saturation.

SOUPS relies on the variable span as an estimate of the cost of firing a transition, but it discounts those portions of the span where we know saturation will not spend much effort. While easy to obtain, the entire variable span of a transition is a rather crude estimate of its impact on performance, while the improvement from taking into account the uniqueness and productivity aspect of a transition is significant, and should be considered a part of any future work on variable ordering.

Many polynomial-time algorithms used to produce orders have in the past targeted graph layout problems and matrix bandwidth minimization. However, saturation and in general decision diagram algorithms have specific characteristics specifically related to the use of computation caches and with no analogue found in those problems. For example, the FORCE and ACCEL algorithms loosely translate graphs and edges into “forces”, but our results would then suggest that modifying these algorithms to somehow consider productivity and uniqueness will be useful, implying the counterintuitive

notion that applying “the same force” multiple times does not make it stronger.

Finally, we conclude with two observations. First, the Petri net formalism provides a well-understood basis for SOUPS, but in no way limits its applicability; the same uniqueness and productivity factors are present in other formalisms and should be utilized whenever possible. Second, our observations about the behavior of saturation led us to the SOUPS metric, but insights can be expected in the other direction as well; specifically, saturation allows for various choices during its execution, and observations of model structure, as is done with SOUPS, can further inform these choices to extract further efficiency.

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation under grant ACI-1642397.

REFERENCES

- [1] Code and data from the paper “SOUPS: a variable ordering metric for the saturation algorithm”. <https://github.com/CycloneMCS/SOUPS>.
- [2] MCC : Model Checking Competition @ Petri Nets. <https://mcc.lip6.fr>.
- [3] F. A. Aloul, I. L. Markov, and K. A. Sakallah. FORCE: a fast and easy-to-implement variable-ordering heuristic. In *Proceedings of the 13th ACM Great Lakes symposium on VLSI*, pages 116–119. ACM, 2003.
- [4] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. Comp.*, 45(9):993–1002, Sept. 1996.
- [5] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In *Proc. TACAS*, LNCS 2031, pages 328–342. Springer, 2001.
- [6] G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In *Proc. TACAS*, LNCS 2619, pages 379–393. Springer, 2003.
- [7] G. Ciardo and A. J. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In *Proc. CHARME*, LNCS 3725, pages 146–161. Springer, 2005.
- [8] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, pages 157–172. ACM, 1969.
- [9] J. Díaz, J. Petit, and M. Serna. A survey of graph layout problems. *ACM Computing Surveys (CSUR)*, 34(3):313–356, 2002.
- [10] C. J. Fisk, D. L. Caskey, and L. E. West. ACCEL: automated circuit card etching layout. *Proceedings of the IEEE*, 55(11):1971–1982, 1967.
- [11] International Organization for Standardization. Petri Net Markup Language (PNML). ISO/IEC 15909-1:2004.
- [12] T. Kam, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.
- [13] E. Keogh, L. Wei, X. Xi, S.-H. Lee, and M. Vlachos. LB_Keogh supports exact indexing of shapes under rotation invariance with arbitrary representations and distance measures. In *Proceedings of the 32nd VLDB Conference*, pages 882–893. VLDB Endowment, 2006.
- [14] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, et al. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
- [15] A. Noack. A ZBDD package for efficient model checking of Petri nets. *Forschungsbericht, Branderburgische Technische Universität Cottbus*, 1999.
- [16] J. Petit. Experiments on the minimum linear arrangement problem. *Journal of Experimental Algorithmics (JEA)*, 8:2–3, 2003.
- [17] J. Petit. Addenda to the survey of layout problems. *Bulletin of EATCS*, 3(105), 2013.
- [18] R. Siminiceanu and G. Ciardo. New metrics for static variable ordering in decision diagrams. In *Proc. TACAS*, LNCS 3920, pages 90–104. Springer, 2006.
- [19] S. Sloan. An algorithm for profile and wavefront reduction of sparse matrices. *International Journal for Numerical Methods in Engineering*, 23(2):239–251, 1986.

Model	Ord	SOT					SOS					SOPS					SOUS					SOUPS				
		R	Min	Avg	Max	Tie	R	Min	Avg	Max	Tie	R	Min	Avg	Max	Tie	R	Min	Avg	Max	Tie	R	Min	Avg	Max	Tie
sg-2-1-2	222	0.27	127228	127228	127228	1	0.59	47585	47585	47585	1	0.59	47585	47585	47585	1	0.55	62257	62257	62257	1	0.55	62257	62257	62257	1
afcs 1 b	500	0.14	26779	30174	33569	2	0.40	24550	24550	24550	1	0.40	24550	24550	24550	1	0.43	24550	24550	24550	1	0.43	24550	24550	24550	1
open system 0	462	0.27	78634	78634	78634	1	0.52	21044	21044	21044	1	0.48	35711	35711	35711	1	0.64	23140	23140	23140	1	0.55	23140	23140	23140	1
ClientServ-2-0	500	0.54	9309	9309	9309	1	0.66	6287	20943	47719	166	0.66	6287	20943	47719	166	0.66	6287	20943	47719	166	0.66	6287	20943	47719	166
tcp5	500	0.22	41337	44950	48563	2	0.64	20908	37950	49379	5	0.64	20908	37950	49379	5	0.77	19757	20586	21414	2	0.77	19757	20586	21414	2
SwimmingPool-2	405	-0.15	7307	15499	23984	53	0.18	4694	16734	48946	265	0.18	4694	16734	48946	265	0.18	4694	16734	48946	265	0.18	4694	16734	48946	265
SafeBus-3	500	0.33	12653	14107	16802	3	0.50	20091	20091	20091	1	0.52	16196	16196	16196	1	0.09	18082	18082	18082	1	0.22	13393	13393	13393	1
ring	450	0.15	40205	40205	40205	1	0.37	7552	7552	7552	1	0.56	25985	25985	25985	1	0.51	27285	27285	27285	1	0.61	11594	11594	11594	1
Peterson-2	500	-0.75	74425	74425	74425	1	-0.15	24906	24906	24906	1	0.38	26941	26941	26941	1	0.49	31181	31181	31181	1	0.68	8290	8290	8290	1
des 01 a	500	0.15	8903	8903	8903	1	0.49	10343	10343	10343	1	0.49	10343	10343	10343	1	0.55	8173	8173	8173	1	0.55	8173	8173	8173	1
trg 1-20-0	500	-0.45	33884	68379	104726	44	0.58	2416	13227	38059	43	0.58	2416	13227	38059	43	0.74	1691	7502	13328	76	0.74	1691	7502	13328	76
closed sys1	489	0.17	9861	9861	9861	1	0.57	6224	6224	6224	1	0.54	6224	6224	6224	1	0.65	5801	5801	5801	1	0.61	5801	5801	5801	1
IBM703	500	0.63	4905	4905	4905	1	0.24	5794	5794	5794	1	0.24	5794	5794	5794	1	0.17	5794	5794	5794	1	0.17	5794	5794	5794	1
2-10 phaseVar	491	0.09	2807	3530	6297	98	0.13	3131	3874	5151	94	-0.57	4917	6772	12171	74	0.62	2878	3616	6573	170	0.59	2878	3616	6573	170
IBM5964	500	-0.68	15450	15450	15450	1	0.78	3453	3453	3453	1	0.78	3453	3453	3453	1	0.80	3583	3583	3583	1	0.80	3583	3583	3583	1
BrVeh-V4-P5-N2	500	0.54	3382	3704	4103	7	0.39	2612	2692	2771	2	0.44	4404	5857	7873	6	-0.34	4793	4990	5102	4	-0.28	2949	3229	3539	3
2D8 grad 10x10 10	438	0.77	3207	3207	3207	1	0.21	3157	3157	3157	1	0.21	3157	3157	3157	1	0.06	3143	3143	3143	1	0.06	3143	3143	3143	1
hgx 110	500	-0.69	25380	38162	53124	74	0.44	3163	4842	6355	9	0.44	3163	4842	6355	9	0.89	2677	2677	2677	1	0.89	2677	2677	2677	1
neoelection-2	278	-0.66	2621	2621	2621	1	-0.28	2688	2688	2688	1	-0.28	2688	2688	2688	1	0.66	1892	1892	1892	1	0.68	1946	1946	1946	1
QCertifProtocol 2	500	-0.23	5118	5118	5118	1	0.27	3573	3651	3729	2	0.27	3573	3651	3729	2	0.82	1941	1941	1941	1	0.82	1941	1941	1941	1
dekker-10	485	0.80	4544	5204	5961	68	0.76	1857	1924	1997	7	0.80	1441	1521	1600	2	-0.10	2189	2189	2189	1	-0.01	1905	1905	1905	1
HouseConst-5	500	0.20	2246	2724	3251	5	0.68	1806	1806	1806	1	0.68	1806	1806	1806	1	0.68	1806	1806	1806	1	0.68	1806	1806	1806	1
dnawalk-02	488	-0.60	4441	7767	11755	100	0.63	1571	2069	2654	47	0.63	1571	2069	2654	47	0.90	1479	1652	1757	68	0.90	1479	1652	1757	68
lampport fmea-2	500	0.32	1634	1634	1634	1	0.44	1271	1271	1271	1	0.59	1612	1612	1612	1	0.39	1993	1993	1993	1	0.49	1612	1612	1612	1
ht d2k1p8b00	480	-0.19	1880	2186	2486	67	0.17	2309	2341	2372	137	0.17	2309	2341	2372	137	0.28	1271	1539	2306	74	0.28	1271	1539	2306	74
deploy 2 a	476	-0.34	12677	12677	12677	1	-0.17	2609	2609	2609	1	0.15	1727	1727	1727	1	0.46	3329	3329	3329	1	0.74	1462	1462	1462	1
G-PPP-1-1	500	0.29	1970	1970	1970	1	0.36	1216	1216	1216	1	0.36	1216	1216	1216	1	0.38	1216	1216	1216	1	0.38	1216	1216	1216	1
philo dyn-3	500	-0.35	2471	2853	3310	78	0.40	1153	1217	1282	10	0.33	1093	1121	1157	3	0.49	1602	1658	1711	6	0.67	1016	1021	1025	2
cs repetitions-2	500	0.71	905	938	971	2	0.13	1622	1671	1700	9	-0.15	1499	1656	2489	13	0.72	898	953	1032	19	0.40	785	993	1207	53
railroad5	500	-0.56	3871	3871	3871	1	-0.00	2094	2094	2094	1	0.33	1505	1505	1505	1	0.83	918	918	918	1	0.85	918	918	918	1
parking 1 4	500	0.46	720	720	720	1	-0.06	1137	1137	1137	1	0.19	662	662	662	1	-0.10	1146	1146	1146	1	0.27	844	844	844	1
TokenRing-5	500	0.48	888	925	969	22	0.68	744	831	956	48	0.57	758	795	852	45	0.54	749	771	794	3	0.54	758	795	852	45
simple lbs-2	500	-0.64	2559	2647	2802	12	0.40	684	684	684	1	0.44	756	760	763	2	0.77	683	683	683	1	0.81	712	712	712	1
AirplaneLD-10	500	-0.37	3112	3112	3112	1	-0.03	2659	2659	2659	1	0.24	788	788	788	1	0.01	1340	1340	1340	1	0.65	693	693	693	1
SmallOS-MT32DC8	333	0.49	411	823	1536	102	0.01	1559	2564	3170	124	0.01	1559	2564	3170	124	0.63	411	615	825	55	0.63	411	615	825	55
MAPK-8	493	0.14	850	1037	1386	3	0.57	502	606	901	18	0.57	502	606	901	18	0.67	489	498	503	19	0.67	489	498	503	19
raft 02	500	-0.29	1234	1234	1234	1	0.45	733	733	733	1	0.13	626	626	626	1	0.69	448	448	448	1	0.65	473	473	473	1
erk-000010	338	0.46	212	529	1355	147	0.79	212	470	760	113	0.79	212	470	760	113	0.82	212	470	760	113	0.82	212	470	760	113
dlcro 03 a	403	-0.04	8576	8576	8576	1	0.04	587	587	587	1	0.36	444	444	444	1	0.17	857	857	857	1	0.27	454	454	454	1
Kanban-5	500	-0.40	475	3223	9178	98	0.84	149	400	617	9	0.84	149	400	617	9	0.85	149	424	588	8	0.85	149	424	588	8
FMS-5	500	-0.02	661	870	1101	3	0.59	391	446	499	6	0.62	432	433	433	3	0.62	352	416	433	5	0.65	352	416	433	5
circadian clock-10	483	0.86	138	142	146	16	0.08	130	696	1202	55	0.05	398	402	407	15	0.26	714	860	992	27	0.37	287	379	407	19
angiogenesis-01	500	0.46	386	386	386	1	0.09	618	618	618	1	0.09	618	618	618	1	0.21	376	376	376	1	0.21	376	376	376	1
Philosophers-10	500	0.32	401	401	401	1	0.75	373	426	494	69	0.75	373	426	494	69	0.79	363	371	374	15	0.79	363	371	374	15
shared mem-5	500	-0.06	487	487	487	1	0.11	432	455	490	8	0.13	425	437	447	5	0.72	352	352	352	1	0.75	353	357	360	2
CircularTrain-12	500	0.48	326	366	414	37	0.98	290	320	350	360	0.98	290	320	350	360	0.98	290	320	350	360	0.98	290	320	350	360
flexbar 04 a	500	0.13	803	803	803	1	-0.53	1740	1740	1740	1	0.55	307	307	307	1	0.76	342	342	342	1	0.80	307	307	307	1
z 2d 3n 1m c 1 2	481	0.83	192	203	211	178	-0.63	211	228	252	188	-0.63	211	228	252	188	0.33	192	209	220	236	0.33	192	209	220	236
rwmutex-r10w10	500	0.73	189	196	209	40	-0.55	624	624	624	1	-0.55	624	624	624	1	0.84	189	191	191	6	0.84	189	191	191	6
database2	500	0.54	185	185	185	1	0.84	169	170	171	3	0.84	169	170	171	3	0.85	173	173	173	1	0.85	173	173	173	1
robot-manip-1	499	0.08	135	157	184	19	0.74	132	151	176	158	0.74	132	151	176	158	0.76	132	151	176	158	0.76	132	151	176	158
joinFree-3	466	-																								